Data and File Structures

(MCA-102)

Unit – 1

[Array (Searching and Sorting), Linked List, Stack and Queue]

by

Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

Introduction

- Data Type: A *data type* is a term which refers to the kinds of data that variables may "hold" in a programming language.
  - For example, a variable of type boolean can assume either the value true or the value false, but no other value.
- Data Structure: A data structure is an arrangement of data in a computer's memory (or sometimes on a disk).
  - In other words, a data structure is meant to be an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.
  - Algorithms manipulate the data in these structures in various ways, such as inserting a new data item, searching for a particular item, or sorting the items.

Categories of Data Structures

- **Linear Data Structures**
  - A data structure whose elements form a sequence, and every element in the structure has a unique **predecessor** and unique **successor**.
  - Examples: **Array**, **Stack**, **Queue**, **Linked List**
- **Non-Linear Data Structures**
  - A data structure whose elements do not form a sequence, there is no unique predecessor or unique successor.
  - Examples: **Tree**, **Graph**

Common Operations on Data Structures

- Traversal: accessing or visiting each data item exactly once
- Searching: finding the data item within the data structure which satisfies searching condition
- Insertion: adding a new data element within the data structure
- Deletion: removing a new data element from the data structure
- Sorting: arranging the data in some logical order
- Merging: combining the data elements of two data structures

Array

- An array is a fixed-size sequential collection of elements of same data type.
- An array is simply a grouping of like-type data.
- In its simplest form, an array can be used to represent a list of numbers, or a list of names.
- Some examples where the concept of an array can be used:
  - List of temperatures recorded every hour in a day
  - List of employees in an organization
  - Test scores of a class of students
  - Table of daily rainfall data
  - etc.

One-Dimensional Array

- A list of items can be given one variable name using only one subscript and such a variable is called a single-scripted variable of a one-dimensional array.
- Declaration:

```
data-type variable-name[size];
```

- Declaration Examples:

```
float height[50];
int group[10];
char name[10];
```


One-Dimensional Array

- Initialization at Compile Time:

```
data-type variable-name[size] = {list of values};
```

- Compile Time Initialization Examples

```
int number[3] = {5, 6, 7};

int age[5] = {22, 24, 23};
            --> Remaining two elements will be initialized to 0.

int counter[] = {1, 2, 3, 4, 5};
            --> The array size may be omitted.

char city[5] = {'D', 'E', 'L'};
            --> Remaining two elements will be initialized to NULL.
```

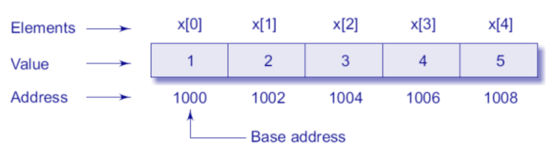

One-Dimensional Array

- Run Time Initialization Examples

```
int counter[10];
for(i=1, i<=10, i++)
{
    counter[i] = i;
}
```

```
Using scanf() function
int counter[10];
for(i=1, i<=10, i++)
{
    scanf("%d", &counter[i]);
}
```


Memory Layout of 1D Array



Calculating Address of Elements in 1D Array

- Let `x[n]` be an one-dimensional array having `n` elements with indices `i` = 0, 1, … …, n-1.
- Then, the address of i^{th} element (`x[i]`) is calculated as follows:

Base Address + (i × Scale Factor of Data Type of Array)

Example: Given an array x[5] of integers with base address = 1000. Calculate the address of element x[3].

Address of x[3] = Base Address + (3 × Scale Factor of Integer)

= 1000 + (3 × 2) = 1006

Two-Dimensional Array

- Declaration:

```
data-type variable-name[row-size] [column-size];
```

- Declaration Examples:

```
float sales[3][3];
int matrix[4][3];
```


Two-Dimensional Array

- Initialization at Compile Time:

```
data-type variable-name[row-size][column-size] = {list of values};
```

- Compile Time Initialization Examples

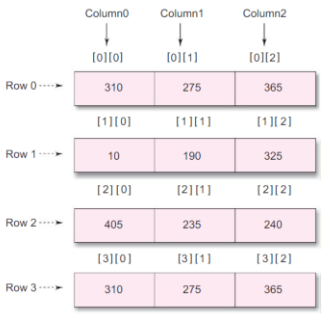
```
int table[2][3] = {1, 1, 1, 2, 2, 2};

int table[2][3] = {{1, 1, 1}, {2, 2, 2}};

--> When array is initialized with all values, explicitly, we need
not specify the size of first dimension.
int table[][3] = {{1, 1, 1}, {2, 2, 2}};

int table[2][3] = {{1, 1}, {2}};
--> It will initialize the first two elements of first row to one,
the first element of second row to two, and all other to zero.
```

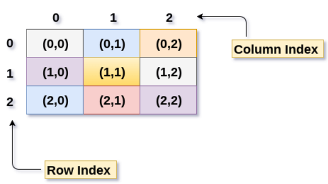
Representation of 2D Array



Memory Layout of 2D Array

- There are two main techniques of storing 2D array elements into memory:
  - Row Major Ordering
    - All the **rows** of the 2D array are stored into the memory contiguously.
  - Column Major Ordering
    - All the **columns** of the 2D array are stored into the memory contiguously.

Memory Layout of 2D Array (contd…)



Row Major Ordering

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

Column Major Ordering

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

Calculating Address of Elements in 2D Array

- Let x[m][n] be a two-dimensional array having m rows and n columns with indices i = 0, 1, … …, m; j = 0, 1, … …n.
- Then, the address of an element x[i][j] of the array, stored in Row Major, is calculated as:

  Base Address + (i × n + j) × Scale Factor of Data Type of Array

Example: Given an array x[5][7] of integers with base address = 900. Calculate the address of element x[4][6].

Address of x[4][6] = 900 + (4 × 7 + 6) × 2 = **968**

Question: Given an array [1…5, 1…7] of integers with base address = 900. Calculate address of element [4, 4].

 U1.16

Calculating Address of Elements in 2D Array

- Let x[m][n] be a two-dimensional array having m rows and n columns with indices i = 0, 1, … …, m; j = 0, 1, … …n.
- Then, the address of an element x[i][j] of the array, stored in Column Major, is calculated as:

  Base Address + (j × m + i) × Scale Factor of Data Type of Array

Example: Given an array x[5][7] of integers with base address = 900. Calculate the address of element x[4][6].

Address of x[4][5] = 900 + (5 × 5 + 4) × 2 = **958**

Question: Given an array [1…5, 1…6] of integers with base address = 2000. Calculate address of element [4, 4].

 U1.17

Sparse Matrix

- A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m×n matrix.
- Sparse matrices are those matrices that have the majority of their elements equal to zero.
  - In other words, the sparse matrix is a matrix that has a greater number of zero elements than the non-zero elements.

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

 U1.18

Limitations of Sparse Matrix

- Storage
  - We need to store m×n (all elements) elements of matric even though maximum number of elements of the matrix are zero.
- Computing Time
  - In case of searching (or performing any operation) in a sparse matrix, we need to traverse m×n (all elements) rather than accessing non-zero elements of the sparse matrix.

Sparse Matrix Representation

- The non-zero elements can be stored with triples, i.e., rows, columns, and value.
- The sparse matrix can be represented in the following ways:
  - Array Representation
  - Linked List Representation
  - List of Lists Representation

Sparse Matrix: Triples/Array Representation)

- A 2D array with 3 row or columns is used to represent the sparse matrix:
  - **Row**: It is an index of a row where a non-zero element is located.
  - **Column**: It is an index of the column where a non-zero element is located.
  - **Value**: The value of the non-zero element is located at the index (row, column).

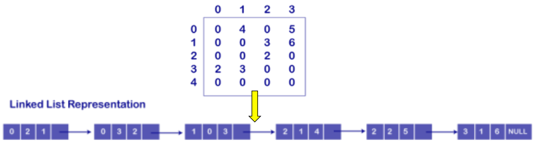
	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

→

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
3	1	3

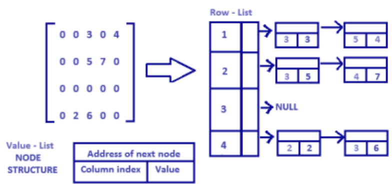
Sparse Matrix: Linked List Representation

- A linear linked list is used to represent the sparse matrix. Each node of the list consists of four fields:
  - **Row**: Row: An index of row where a non-zero element is located.
  - **Column**: An index of column where a non-zero element is located.
  - **Value**: Value of the non-zero element which is located at the index (row, column).
  - **Next Node**: It stores the address of the next node.



Sparse Matrix: List of List Representation

- One list is used to represent the rows, and each row contains the list of triples:
  - **Column**: An index of column where a non-zero element is located.
  - **Value**: Value of the non-zero element.
  - **Address of Next Node** : It stores the address of the next non-zero element.



Linear Search

- Searching is a process of finding a value in a list of values.
- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

Linear Search: Step-by-Step Process

- **Step 1:** Read the element to be searched from the user
- **Step 2:** Compare, the element to be searched with the first element in the list.
- **Step 3:** If both are matched, then display "Given element found" and terminate the search process.
- **Step 4:** If both are not matched, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also not matched, then display "Element not found!" and terminate the function.

Linear Search: Working Example

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Linear Search: Working Example

Step 2:

search element (12) is compared with next element (20)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Linear Search: Working Example

Step 4:

search element (12) is compared with next element (55)

list	0	1	2	3	4	5	6	7
	65	20	10	55	32	12	50	99
				12				

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list	0	1	2	3	4	5	6	7
	65	20	10	55	32	12	50	99
					12			

Both are not matching. So move to next element

Linear Search: Working Example

Step 6:

search element (12) is compared with next element (12)

list	0	1	2	3	4	5	6	7
	65	20	10	55	32	12	50	99
						12		

Both are matching. So we stop comparing and display element found at index 5.

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly, the data collection should be in the sorted form.

Binary Search: Step-by-Step Process

- **Step 1:** Read the element to be searched from the user.
- **Step 2:** Find the middle element in the sorted list.
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matched, then display "Given element found!" and terminate the search process.
- **Step 5:** If both are not matched, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub-list of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub-list of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until the sub-list contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!" and terminate the function.

Binary Search: Working Example

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element **12**

Step 1:

search element (12) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					12				

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Binary Search: Working Example

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
		12							

Both are matching. So the result is "Element found at index 1"

Program Code for Binary Search

```
low=0;
high=n-1;
while(low<=high) {
        mid=(low+high)/2;
        if(item<a[mid])
                high=mid-1;
        else if(item>a[mid])
                low=mid+1;
        else if(item==a[mid]) {
                printf("Item Found");
                break;
        }
        else {
                printf("Not Found");
        }
}
```


Selection Sort: Step-by-Step Process

- **Step 1:** Select the first element of the list (i.e., element at first position in the list).
- **Step 2:** Compare the selected element with all other elements in the list.
- **Step 3:** For every comparison, if any element is smaller than selected element (for ascending order), then these two are swapped.
- **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

- Complexity: $O(n^2)$

Selection Sort: Working Example

- Consider the following unsorted list of elements:

15	20	10	30	50	18	5	45

Iteration 1:

Select the **first** element of the list,

- compare it with all other elements in the list, and
- whenever we found a smaller element than the element at first position then swap those two elements.

15	20	10	30	50	18	5	45

15 > 20
FALSE

15	20	10	30	50	18	5	45

15 > 10
TRUE
SWAP

Selection Sort: Working Example

10	20	15	30	50	18	5	45

10 > 30
FALSE

10	20	15	30	50	18	5	45

10 > 50
FALSE

10	20	15	30	50	18	5	45

10 > 18
FALSE

10	20	15	30	50	18	5	45

10 > 5
TRUE
SWAP

5	20	15	30	50	18	10	45

5 > 45
FALSE

List after first iteration

5	20	15	30	50	18	10	45

Selection Sort: Working Example

Iteration 2:

Select the **second** position element of the list,

- compare it with all other elements in the list, and
- whenever we found a smaller element than the element at second position then swap those two elements.

List after second iteration

5	10	20	30	50	18	15	45

Iteration 3:

List after third iteration

5	10	15	30	50	20	18	45

Selection Sort: Working Example

List after fourth iteration

5	10	15	18	50	30	20	45

List after fifth iteration

5	10	15	18	20	50	30	45

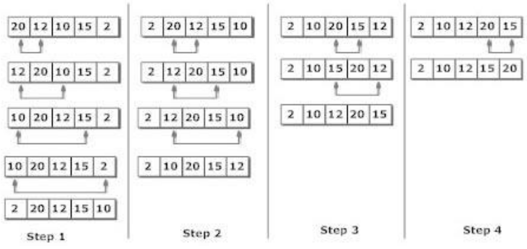
List after sixth iteration

5	10	15	18	20	30	50	45

List after seventh iteration

5	10	15	18	20	30	45	50

Selection Sort: Working Example



Program Code for Selection Sort

```
for(i=0; i<size; i++)
{
    for(j=i+1; j<size; j++)
    {
        if(list[i] > list[j])
        {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
        }
    }
}
```


Revised Program Code for Selection Sort

```
for(i=0; i<n-1; i++)
{
    int min = i; //Consider the first element as minimum.
    for(j=i+1; j<n; j++)
    {
        if(a[j] < a[min])
        {
            min = j;
        }
    }
    if(min != i)
    {
        swap(a[i], a[min]);
    }
}
```

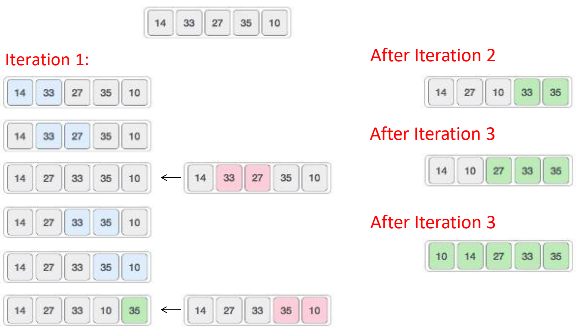
Bubble Sort: Step-by-Step Process

- **Step 1:** Select the first element of the list (i.e., element at first position in the list).
- **Step 2:** Compare the current element with next element of the list.
- **Step 3:** If the current element is greater than the next element (for ascending order), then these two are swapped.
- **Step 4:** If the current element is less than the next element, move to the next element.
- **Step 5:** Repeat from Step 1.

- Complexity: $O(n^2)$

Bubble Sort: Working Example

- Consider the following unsorted list of elements:



Program Code for Bubble Sort

```
for(i=1;i<n;i++) {
      for(j=0;j<n-i;j++) {
            if(a[j]>a[j+1])
            {
                  temp=a[j];
                  a[j]=a[j+1];
                  a[j+1]=temp;
            }
      }
}
```


Insertion Sort: Step-by-Step Process

- **Step 1:** Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

- Complexity: $O(n^2)$

Insertion Sort: Working Example

- Consider the following unsorted list of elements:

15	20	10	30	50	18	5	45

- Assume that the sorted portion of the list is empty and all elements in list are in unsorted portion, as shown below:

Sorted | Unsorted

15	20	10	30	50	18	5	45

- Move the first element **15** from the unsorted portion to sorted portion.

Sorted | Unsorted

15	20	10	30	50	18	5	45

Insertion Sort: Working Example

- To move **20** from unsorted to sorted portion, compare **20** with **15** and insert it at correct position.

Sorted | Unsorted

15	20	10	30	50	18	5	45

- To move element **10** from unsorted portion to sorted portion, compare **10** with **20**, it is smaller so perform swapping. Then, compare **10** with **15**, again it is smaller so perform swapping.

Sorted | Unsorted

10	15	20	30	50	18	5	45

Insertion Sort: Working Example

- Similarly, an element from unsorted portion is retrieved and is compared with element in sorted portion and is inserted accordingly.

Sorted | Unsorted

10	15	20	30	50	18	5	45

Sorted | Unsorted

10	15	20	30	50	18	5	45

Sorted | Unsorted

10	15	18	20	30	50	5	45

 U1.49

Insertion Sort: Working Example

Sorted | Unsorted

5	10	15	18	20	30	50	45

Final Sorted List

5	10	15	18	20	30	45	50

 U1.50

Insertion Sort: Working Example

Step 1	12 3 1 5 8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3 12 1 5 8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1 3 12 5 8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1 3 5 12 8	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.

 U1.51

Program Code for Insertion Sort

```
for(i=1; i<n; i++)
{
    temp = data[i];
    j = i-1;
    while(temp<data[j] && j>=0)
    {
        data[j+1] = data[j];
        j = j-1;
    }
    data[j+1]=temp;
}
```


Shell Sort

- In Insertion Sort, a large number of swaps/shifts are performed to sort the elements.
- Shell sort is an efficient sorting algorithm and is based on Insertion Sort.
- This algorithm avoids large shifts as in case of Insertion Sort, if the smaller value is to the far right and has to be moved to the far left.
- Shell Sort compares items that lie far apart which allows elements to move faster to the front of the list.

Shell Sort: Algorithm Working

1. Divide the list into sub-lists using interval **Floor(N/2^k)**.
   - Shell Sequence **(Floor(N/2^k))**
2. Short sub-lists using Insertion Sort.
3. Repeat until complete list is sorted.

Shell Sort: Working

- Let the list of elements be: **35**, **33**, **42**, **10**, **14**, **19**, **27**, **44**
- Gap/Interval = Floor($8/2^1$) = **4**
- Sub-lists: {35, 14}, {33, 19}, {42, 27}, and {10, 44}
- Short sub-lists using Insertion Sort.

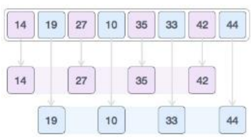
After this step, array becomes

14	19	27	10	35	33	42	44

 U1.55

Shell Sort: Working

- In next phase, the interval becomes Floor($8/2^2$) = **2**
- Then, we take interval of 2 and this gap generates two sub-lists {14, 27, 35, 42} and {19, 10, 33, 44}.
- Short sub-lists using Insertion Sort.



After this step, array becomes
14, 10, 27, 19, 35, 33, 42, 44

 U1.56

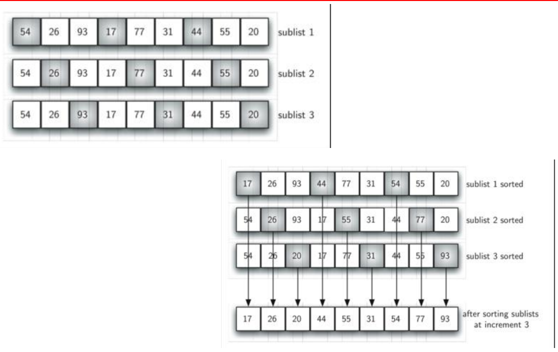
Shell Sort: Working

- In next phase, the interval becomes Floor($8/2^3$) = **1**
- Finally, sort (using Insertion Sort) the rest of the array using interval of value 1.

10	14	19	27	33	35	42	44

 U1.57

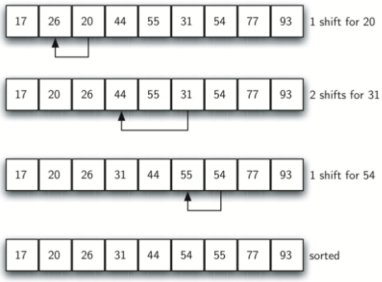
Shell Sort: Example



 U1.58

Shell Sort: Example

Sorting by using interval of value 1 (using Insertion Sort)



 U1.59

Program Code for Shell Sort

```
for(gap = n/2; gap >= 1; gap = gap/2) {
        for(j = gap; j < n ; j++) {
                for(i = j-gap; i >= 0; i = i - gap) {
                        if(a[i+gap] > a[i]) {
                                break;
                        }
                        else {
                                swap(a[i+gap], a[i])
                        }
                }
        }
}
```

 U1.60

Radix Sort

- A list of numbers is sorted based on the digits of individual numbers.
- Sorting is performed from least significant digit to the most significant digit.
- The number of passes required are equal to the number of digits present in the largest number of the list.
  - Example: If the largest number has 3 digits, then the list will be sorted in 3 passes.

Radix Sort: Algorithm

1. Define 10 queues, each representing a bucket for each digit from 0 to 9.
2. Consider the least significant digit of each number in the list which is to be sorted.
3. Insert each number into their respective queue based on the least significant digit.
4. Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
5. Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Radix Sort: Example

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Radix Sort: Example

Pass - 1

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 100, 12, 150, 77, 55 & 23

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

 U1.64

Radix Sort: Example

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

 U1.65

Radix Sort: Example

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

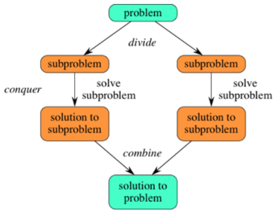
12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

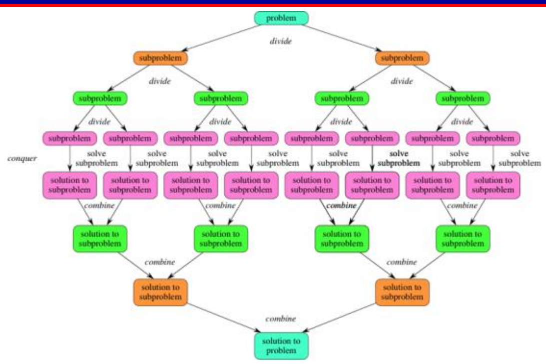
 U1.66

Divide and Conquer

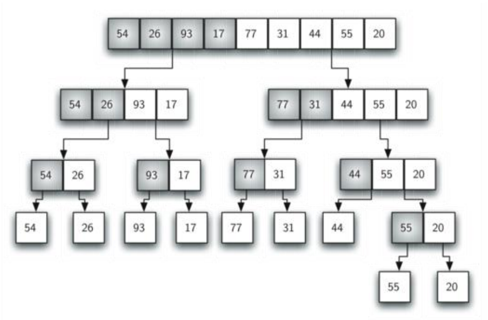
1. **Divide** the problem into multiple small problems.
2. **Conquer** the sub-problems by solving them. The idea is to break down the problem into atomic sub-problems, where they are actually solved.
3. **Combine** the solutions of the sub-problems to find the solution of the actual problem.



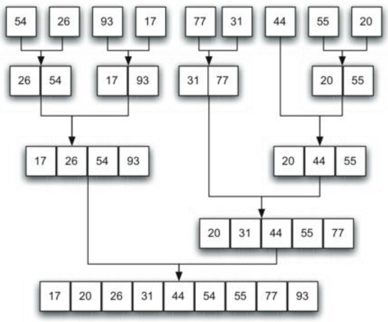
Divide and Conquer



Merge Sort: Working



Merge Sort: Working



Algorithm/Code for Merge Sort

```
mergeSort(list, lower, upper)
{
    if(lower < upper)
    {
        mid = (lower + upper)/2;
        mergeSort(list, lower, mid)
        mergeSort(list, mid+1, upper)
        merge(list, lower, mid, upper)
    }
}
```


Code for Merging in Merge Sort

```
merge(a, lower, mid, upper)
{
    int i, j, k, b[n];
    i = lower;
    j = mid+1;
    k = lower;
    while (i <= mid && j <= upper)
    {
        if (a[i] <= a[j])
        {
            b[k] = a[i];
            i++; k++;
        }
        else
        {
            b[k] = a[j];
            j++; k++;
        }
    }
    //continued … … …
```

```
//If any element is left in sub-lists
    if (i > mid)
    {
        while (j <= upper)
        {
            b[k] = a[j];
            j++;
            k++;
        }
    else
    {
        while (i <= mid)
        {
            b[k] = a[i];
            i++;
            k++;
        }
    }
//We may copy the items to main array
}
```

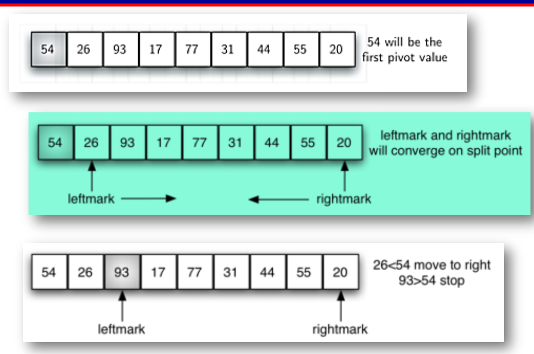
Quick Sort: Procedure/Process

- The quick sort uses **divide and conquer** to gain the same advantages as the merge sort, while not using additional storage.
- A quick sort first selects a value, which is called the pivot value. The **actual position** where the pivot value belongs in the final sorted list, commonly called the **split point**, is used to divide the list for subsequent calls to the quick sort.
- Partitioning begins by locating two position markers—let's call them leftmark and rightmark — **at the beginning** and **end of the remaining items in the list**.

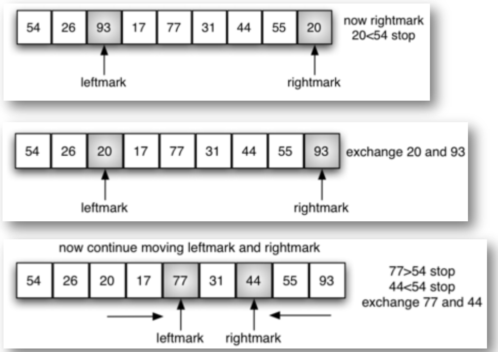
Quick Sort: Procedure/Process

- Begin by incrementing leftmark until we locate a value that is greater than the pivot value.
- Then decrement rightmark until we find a value that is less than the pivot value.
- At the point where rightmark becomes less than leftmark, we stop.
  - The position of rightmark is now the split point.
  - The pivot value can be exchanged with the contents of the split point and the pivot value is now in place.

Quick Sort: Working

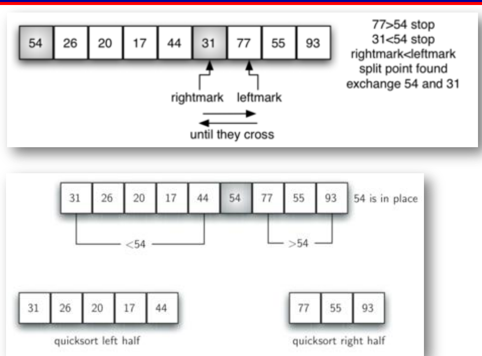


Quick Sort: Working



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U1.76

Quick Sort: Working



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U1.77

Algorithm/Code for Quick Sort

```
quickSort(list, low, high)
{
    int pivot;
    if ( high > low ) //Termination Condition
    {
        pivot = partition(a, low, high);
        quickSort(a, low, pivot-1);
        quickSort(a, pivot+1, high);
    }
}
```

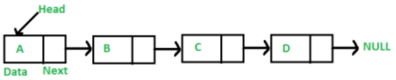
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U1.78

Linked List

- Linked List is a linear collection of data elements, called *nodes*.
- The linear order is given by pointers.
- Each node is divided into two or more parts.
- A Linked List can be of following types:
  - Linear Linked List (One-Way List)
  - Doubly Linked List (Two-Way List)
  - Circular Linked List

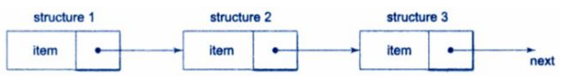
Linear Linked List

- **Linked List** is a linear data structure which consists of a series of nodes.
- Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



- **Advantages:**
  - **Dynamic data structure**: can grow or shrink dynamically
  - **Ease of insertion/deletion**: insertion and deletion are efficient
  - **Implementation of other complex data structures**
- **Drawbacks:**
  - **No random access**: access to an arbitrary data item is time-consuming
  - **Requires more memory**: extra space is required for pointer

Implementation of Linear Linked List




```
//Structure Representation for Node of a Linear Linked List
struct node
{
    int item;
    struct node *next;
};
```

Insertion of a Node in Linear Linked List

```
//Insertion of a Node in the Beginning of a Linear Linked List
void insertBegin(int item) {
 NODE *node;
 node=(NODE*)malloc(sizeof(NODE));
 node->data=item;
 if(start==NULL) {
   node->next=NULL;
 }
 else {
   node->next=start;
 }
 start=node;
}
```


Insertion of a Node in Linear Linked List

```
//Insertion of a Node in the End of a Linear Linked List
void insertEnd(int item) {
  NODE *node,*pos;
  node=(NODE*)malloc(sizeof(NODE));
  node->data=item;
  node->next=NULL;
  if(start==NULL) {
    start=node;
  }
  else {
    pos=start;
    while(pos->next!=NULL) {
      pos=pos->next;
    }
    pos->next=node;
  }
}
```


Insertion of a Node in Linear Linked List

```
//Insertion of a Node at Specific Position of a Linear Linked List
void insertPosition(int item,int p) {
  NODE *node,*pos;
  int count=1;
  pos=start;
  while(count<p)
    if(count==(p-1)) {
      node=(NODE*)malloc(sizeof(NODE));
      node->data=item;
      node->next=pos->next;
      pos->next=node;
      break;
    }
    else {
      pos=pos->next;
      count++;
    }
}
```

Deletion of a Node from Linear Linked List

```
//Deletion of a Node from the Beginning of a Linear Linked List
void deleteBegin() {
   NODE *node;
   if(start==NULL) {
       printf("\nUNDERFLOW");
       return;
   }
   else {
       node=start;
       start=start->next;
       printf("NODE DELETED %d ", node->data);
       free(node);
   }
}
```


Deletion of a Node from Linear Linked List

```
//Deletion of a Node from the End of a Linear Linked List
void deleteEnd() {
    NODE *node,*pos;
    if(start==NULL) {
         printf("\nUNDERFLOW");
         return; }
    else if(start->next==NULL) {
        node=start;
        start=NULL;
        printf("\nNODE DELETED %d", node->data);
        free(node); }
    else {
        pos=start;
        node=start->next;
        while(node->next!=NULL) {
            pos=node;
            node=node->next; }
   pos->next=NULL;
   printf("\nNODE DELETED %d", node->data);
   free(node);
 }
}
```


Deletion of a Node from Linear Linked List

```
//Deletion of a Node from Specific Position of a Linear Linked List
void deletePosition(int p) {
    NODE *node,*pos;
    pos=start;
    int count=0;
    while(count<p) {
        if(count==(p-1)) {
            node=pos->next;
            pos->next=node->next;
            free(node);
            break;
        }
        else {
            pos=pos->next;
            count++;
        }
    }
}
```

Traversal of a Linear Linked List

```
//Traversal of a Linear Linked List
void travers() {
    NODE *pos;
    pos=start;
    if(pos==NULL) {
        printf("\nLIST IS EMPTY");
    }
    else {
        printf("\nLIST ELEMENTS: ");
        while(pos!=NULL) {
            printf("%d  ",pos->data);
           pos=pos->next;
        }
    }
}
```


Polynomials Addition using Linear Linked List

```
void addPoly(NODE **start, NODE *p, NODE *q) {
  NODE *node = (NODE *)malloc(sizeof(NODE));
  node->next = NULL;
  *start = node;
  while(p && q) { //LOOP WHILE BOTH LISTS HAVE VALUES
    if(p->pow > q->pow) {
      node->pow = p->pow;    node->coe = p->coe;    p = p->next;
    }
    else if(p->pow < q->pow) {
      node->pow = q->pow;    node->coe = q->coe;    q = q->next;
    }
    else {
      node->pow = p->pow;    node->coe = p->coe + q->coe;    p = p->next;    q = q->next;
    }
    if(p && q) { //GROW THE LINKED LIST ON CONDITION
       node->next = (NODE *)malloc(sizeof(NODE));
       node = node->next;
       node->next = NULL;
    }
  }
```


Polynomials Addition using Linear Linked List

```
  //continued…
  while(p || q) {
    NODE *newNode = (NODE *)malloc(sizeof(NODE));
    node->next = newNode;
    node = newNode;
    node->next = NULL;
    if(p) {
      node->pow = p->pow;    node->coe = p->coe;    p = p->next;
    }
    if(q) {
      node->pow = q->pow;    node->coe = q->coe;    q = q->next;
    }
  }
}
```


Stack

- A Stack is a linear data structure.
- It is a list in which insertion of new data item and deletion of existing data item is done from one end, known as **Top** of Stack.
- Stack is also called **LIFO** (Last-in-First-out) type of list.
  - The last inserted element will be the first to be deleted from Stack.
- Example:
  - Some of you may eat biscuits (or poppins). If you assume only one side of the cover is torn and biscuits are taken out one by one. This is called **poping**. If you want to preserve some biscuits for some time later, you will put them back into the pack through the same torn end. This is called **pushing**.

Operations on Stack

- Push
  - The process of inserting a new element to the top of stack is called Push operation.
  - In case the list is full, no new element can be accommodated, it is called Stack Overflow condition.
- Pop
  - The process of deleting an element from top of stack is called Pop operation.
  - If there is no any element in the Stack and Pop is performed then this will result in Stack Underflow condition.

Implementation of Stack

- Static Implementation
  - It is achieved using Array
- Dynamic Implementation
  - It is achieved using Linked List

Implementation of Stack using Array

- Push Operation

```
int stack[10],top = -1;
void push(int x)
{
    top = top+1;
    stack[top] = x;
}
```

- Pop Operation

```
int pop()
{
    int temp;
    temp = stack[top];
    top = top-1;
    return temp;
}
```

 U1.94

Implementation of Stack using Linked List

- Structure Definition

```
struct stack
{
    int data;
    struct node *next;
};
typedef struct stack STACK;
STACK *top;
```

- Required Functions

```
void create();
int isempty();
int isfull()
void push(int);
int pop();
void display();
```

 U1.95

Some Applications of Stack

- Reverse of String/Number
- Recursion (Recursive Function)
- Expression Conversion
- Expression Evaluation
- Syntax Parsing
- Undo-mechanism in an Editor
- etc.

 U1.96

Expressions and their Types

- An expression is defined as a number of operands or data items combined using several operators.
- The way to write arithmetic expression is known as a notation.
- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.
- These notations are:
  - Infix Notation
  - Prefix Notation
  - Postfix Notation

Infix Notation

- **Infix Notation** is what we come across in our general mathematics.
- In **Infix Notation**, operators are written *in-between* the operands.
- Example:
  - Expression to add two numbers A and B is written as:

A + B

- **Infix Notation** needs precedence of the operators and we sometimes use bracket () to override these rules.

Prefix Notation

- In **Prefix Notation**, operators are written *before* the operands.
- This is also known as *polish notation* in the honor of the Polar mathematician (Jan Lukasiewicz) who developed this notation.
- Example:
  - Expression to add two numbers A and B is written as:

+ A B

Postfix Notation

- In **Postfix Notation**, operators are written *after* the operands.
- This is also known as *reverse polish notation*.
- Example:
  - Expression to add two numbers A and B is written as:

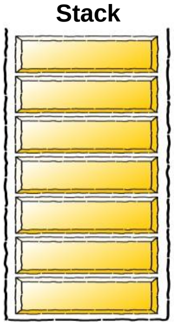
    A B +
- It is most suitable for computer to calculate any expression as there is no need for operator precedence and other rules.
- It is the universally accepted notation for designing ALU of the CPU, **therefore important for us to study**.

Conversion from Infix to Postfix Notation

- While there are tokens to be read from expression, read the token.
- If the token is an operand, then insert it to output.
- If the token is an operator and if the Top of Stack is not any operator then push the operator to stack.
- If the token is an operator O1:
  - While there is an operator, O2 at top of stack (O2 is Top), and
  - If precedence of O1 > O2
    - ✓ Push O1 on to Stack (now O1 is Top)
  - Else if precedence of O1 <= O2
    - ✓ Pop O2 to the output and Push O1 onto Stack
- If the token is a left parenthesis, the Push it onto the Stack.
- If the token is a right parenthesis:
  - Until the token at Top is a left parenthesis, Pop operators off the Stack onto the output
  - Pop the left parenthesis from Stack
- If the token at Top is an operator, Pop and insert it onto output.

Step-by-Step Example: Infix to Postfix Conversion

Stack



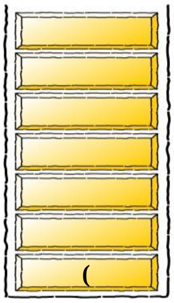
Infix

(a + b - c) * d – (e + f)

Postfix

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

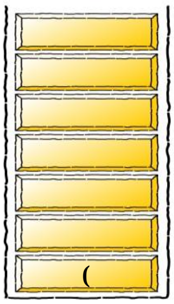
a + b - c) * d – (e + f)

Postfix

 U1.103

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

+ b - c) * d – (e + f)

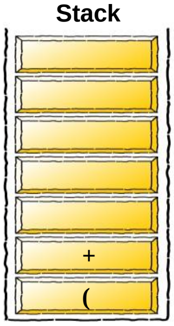
Postfix

a

 U1.104

Step-by-Step Example: Infix to Postfix Conversion

Stack



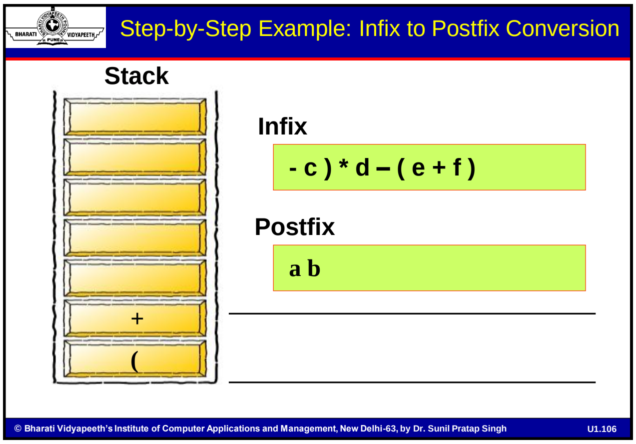
Infix

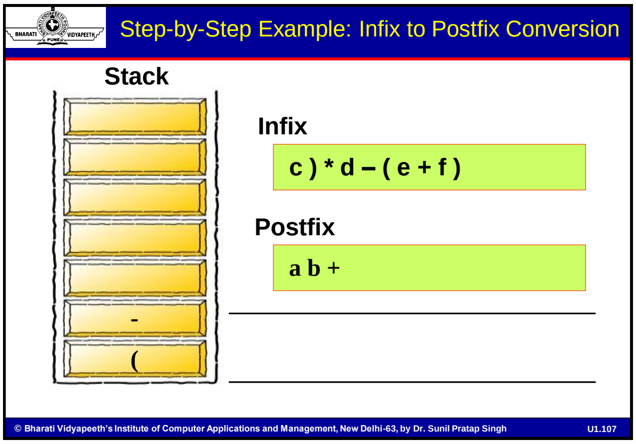
b - c) * d – (e + f)

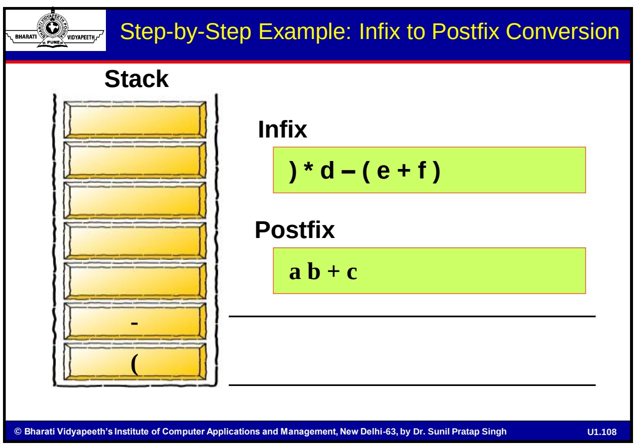
Postfix

a

 U1.105

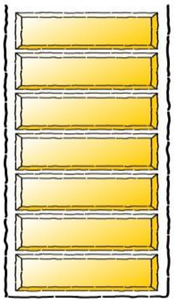

Step-by-Step Example: Infix to Postfix Conversion
Stack
+
(
Infix
- c) * d – (e + f)
Postfix
a b
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U1.106


Step-by-Step Example: Infix to Postfix Conversion
Stack
-
(
Infix
c) * d – (e + f)
Postfix
a b +
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U1.107


Step-by-Step Example: Infix to Postfix Conversion
Stack
-
(
Infix
) * d – (e + f)
Postfix
a b + c
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U1.108

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

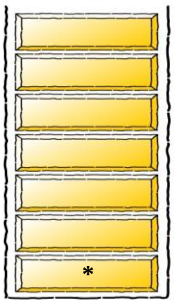
* d – (e + f)

Postfix

a b + c -

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

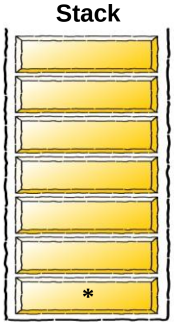
d – (e + f)

Postfix

a b + c -

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

– (e + f)

Postfix

a b + c - d

Step-by-Step Example: Infix to Postfix Conversion

Stack

-

Infix

(e + f)

Postfix

a b + c – d *

 U1.112

Step-by-Step Example: Infix to Postfix Conversion

Stack

(
-

Infix

e + f)

Postfix

a b + c – d *

 U1.113

Step-by-Step Example: Infix to Postfix Conversion

Stack

(
-

Infix

+ f)

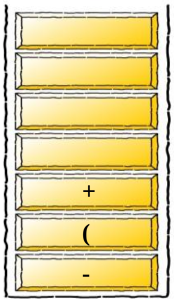
Postfix

a b + c – d * e

 U1.114

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

f)

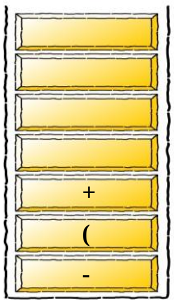
Postfix

a b + c – d * e

 U1.115

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

)

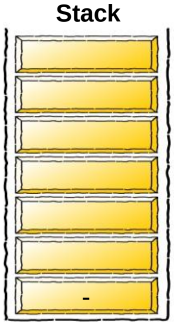
Postfix

a b + c – d * e f

 U1.116

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

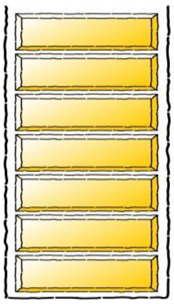
Postfix

a b + c – d * e f +

 U1.117

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

Postfix

a b + c – d * e f + -

Infix to Postfix Conversion: Example

Convert ((A – (B + C)) * D) ^(E + F) to Postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion from Infix to Prefix Notation

- The conversion process is almost same according to Postfix notation.
  - The only change from Postfix form is that **traverse the expression from right to left** and **the operator is placed before the operand** rather than after them.
- Convert the expression A * B + C / D into Prefix notation.
  - Answer: + * A B/ C D

Conversion from Postfix to Infix Notation

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Postfix to Infix Conversion: Example

Convert the expression A B C * D E F ^ / G * - H + * to Infix notation.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) – ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) – ((D/(E^F))*G)) H	Push H
*	A (((B*C) – ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

Infix to Postfix Conversion: Questions

- A * B + C
- A + B * C
- A * (B + C)
- A – B + C
- A * B ^ C + D
- A * (B + C * D) + E
- (A + B) * C / D + E ^ F /G → A B + C * D / E F ^ G / + (Answer)
- A + (B * C – (D / E ^ F) * G) * H → A B C * D E F ^ / G * - H * + (Answer)
- A - B / (C * D ^ E) → A B C D E ^ * / - (Answer)

Evaluation of Postfix Expression

Expression: 4 5 6 * +

Step	Input	Operation	Stack	Calculation
1	4	Push	4	
2	5	Push	4 5	
3	6	Push	4 5 6	
4	*	Pop 2 Elements and Evaluate	4	6 * 5 = 30
5		Push Result (30)	4 30	
6	+	Pop 2 Elements and Evaluate	Empty	4 + 30 = 34
7		Push Result (34)	34	
8		No More Elements (Pop)	Empty	34

Double Stack/Multistack

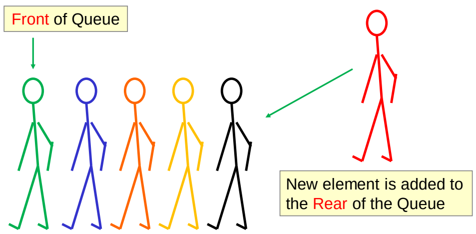
- Double stack means two stacks which are implemented using a single array.
- To prevent memory wastage, the two stacks are grown in opposite direction.
- The pointer Top1 and Top2 points to top-most element of Stack1 and Stack 2 respectively.
- Initially, Top1 is initialized to -1 and Top2 is initialized the size of array.
- As the elements are pushed into Stack1, Top1 is incremented.
- Similarly, as the elements are pushed into Stack2, Top2 is decremented.
- The array is full when Top1=Top2-1.
- Multistack means more than 2 stacks which are implemented using a single array.

Queue

- A Queue is a linear data structure.
- It is a list in which insertion of new data items is done from one end, called **Rear** end, and deletion of existing data item is done from other end, known as **Front** end of Queue.
- Queue is also called **FIFO** (First-in-First-out) type of list.
  - The first inserted element will be the first to be deleted from Queue.

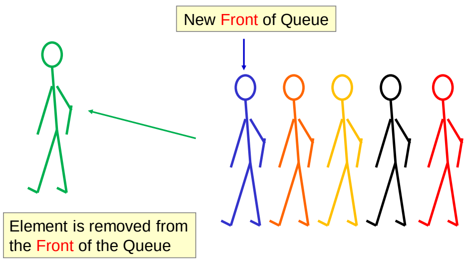
Conceptual View of a Queue

- **Inserting/Adding an Element in Queue**



Conceptual View of a Queue

- **Deleting/Removing an Element from Queue**



Applications of Queue

- Real World Examples
  - People on an Escalator or Waiting in a Line
  - Cars at a Gas Station
- Computer Science Examples
  - Print Queue
  - Keyboard Input Buffer
  - Queue of Network Data Packets
  - Queue of Processes
- Applications in Simulation Studies

Working of Queue

- Enqueue
  - The process of inserting a new element at the Back of queue is called Enqueue operation.
  - In case the list is full, no new element can be accommodated, it is called Queue Overflow condition.
- Dequeue
  - The process of deleting an element from Front of queue is called Dequeue operation.
  - If there is no any element in the queue and Dequeue is performed then this will result in Queue Underflow condition.

Working of Queue

- Empty Queue

F = -1
R = -1

0	1	2	3	4

- Queue after inserting 1 elements

F = 0
R = 0

20				
0	1	2	3	4

- Queue after inserting 2 more elements

F = 0
R = 2

20	30	40		
0	1	2	3	4

Working of Queue

- Queue after deleting 2 elements

F = 2
R = 2

		40		
0	1	2	3	4

- Queue after inserting 2 elements

F = 2
R = 4

		40	50	60
0	1	2	3	4

- What if we want to insert 1 more element?

F = 2
R = 4

		40	50	60
0	1	2	3	4

  - Insertion not possible because **R = 4**.

Implementation of Queue

- Static Implementation
  - It is achieved using Array
- Dynamic Implementation
  - It is achieved using Linked List

 U1.133

Implementation of Queue using Array

- Insertion (Enqueue)

```
#define Max 5
#define Nil -1
int queue[Max];
int front, rear;

void enqueue(int x) {
    if(front == Nil) {
        front = rear = 0;
    }
    else {
        rear = rear + 1;
    }
    queue[rear] = a;
}
```

 U1.134

Implementation of Queue using Array

- Deletion (Dequeue)

```
int dequeue(int x)
{
    int temp = queue[front];
    if(rear == front)
    {
        front = rear = Nil;
    }
    else
    {
        front = front + 1;
    }
    return temp;
}
```

 U1.135

Implementation of Queue using Array

- Traversal (Display/Print Elements)

```
void display()
{
    int i;
    for(i = front; i <= rear; i++)
    {
        printf("%d  ",queue[i]);
    }
}
```


Implementation of Queue using Linked List

- Structure Definition

```
struct queue
{
    int data;
    struct node *next;
};
typedef struct queue QUEUE;
QUEUE *start;
```

- Required Functions

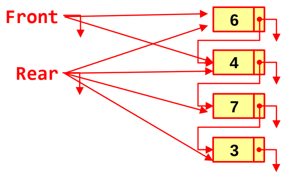
```
void create();
int isempty();
int isfull()
void enqueue(int);
int dequeue();
void display();
```


Implementation of Queue using Linked List

- In Queue, insertion takes place at Rear end.
  - This is similar to inserting an element at the **end** of a Linked List.
- In Queue, deletion takes place at Front end.
  - This is similar to deleting an element from the **front** of a Linked List.
- Therefore, Linked List has application in implementing a Queue.

Queue using Linked List

```
Enqueue(6);
Enqueue(4);
Enqueue(7);
Enqueue(3);
Dequeue();
```



Structure for a Queue using Linked List

```
struct queue
{
    int data;
    struct node *next;
};
typedef struct queue QUEUE;
QUEUE *start;
```


Insertion in a Queue using Linked List

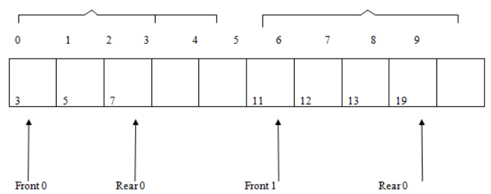
```
void enqueue(int a)
{
    QUEUE *node,*pos;
    node = (QUEUE*)malloc(sizeof(QUEUE));
    node->data = a;
    node->next = NULL;
    if(start == NULL)
        start = node;
    else
    {
        pos = start;
        while(pos->next != NULL)
            pos = pos->next;
        pos->next = node;
    }
}
```

Deletion from a Queue using Linked List

```
int dequeue()
{
    QUEUE *node;
    int item;
    if(start != NULL)
    {
        node = start;
        item = node->data;
        start = start->next;
        free(node);
        return (item);
    }
    else
        return 0;
}
```


Multiqueue

- Maintaining two or more queues in the same array refers to Multiqueue.
- Double Queue:



Limitations of Linear Queue (With Array)

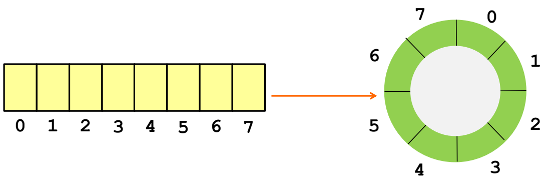
- Consider the following representation of Queue:

F = 2
R = 4

		40	50	60
0	1	2	3	4

- Even after having 2 unoccupied cells, we are unable to insert data elements because insertion is done at Rear end, and Rear is pointing to last position of the Queue.
- Solution?
  - **Circular Queue**

Circular Queue



Circular Queue

- Circular Queue is a linear data structure
- The operations are performed based on FIFO (First In, First Out) principle
- The last position is connected back to the first position to make a circle

Working of Circular Queue

- Empty Queue

F = -1
R = -1

0	1	2	3	4

- Queue after inserting 1 elements

F = 0
R = 0

10				
0	1	2	3	4

- Queue after inserting 4 more elements

F = 0
R = 4

10	**20**	**30**	**40**	**50**
0	1	2	3	4

→ **FULL**

Working of Circular Queue

- Queue after deleting 1 elements

F = 1
R = 4

	20	30	40	50
0	1	2	3	4

- Queue after deleting 1 more elements

F = 2
R = 4

		30	40	50
0	1	2	3	4

- Queue after inserting 2 more element

F = 2
R = 1

60	70	30	40	50
0	1	2	3	4

→ FULL

Implementing Circular Queue using Array

```
//Method 1 to Check Queue Overflow
int isFull() {
    if((front == rear + 1) || (front == 0 && rear == SIZE-1))
        return 1;
    else
        return 0;
}
```

```
//Method 2 to Check Queue Overflow
int isFull() {
    if((rear+1) % SIZE == front)
        return 1;
    else
        return 0;
}
```

```
int isEmpty() {
    if(front == -1)
        return 1;
    else
        return 0;
}
```


Insertion in Circular Queue using Array

```
void insert(int item)
{
    if(isFull())
        printf("OVERFLOW!");
    else
    {
        if(front == -1)
        {
             front = 0;
        }
        rear = (rear + 1) % SIZE;
        queue[rear] = item;
    }
}
```

Deletion in Circular Queue using Array

```
int delete() {
    int item;
    if(isEmpty()) {
        printf("UNDERFLOW!");
        return(-1);
    }
    else {
        item = queue[front];
        if(front == rear) {
            front = -1;
            rear = -1;
        }
        else {
            front = (front + 1) % SIZE;
        }
        printf("ITEM DELETD %d", item);
        return (element);
    }
}
```


Traversal of a Circular Queue using Array

```
int travers()
{
    int i;
    if(isEmpty())
        printf("UNDERFLOW!");
    else
    {
         printf("ITEMS: ");
        for(i = front; i!=rear; i=(i+1)%SIZE) {
            printf("%d ",queue[i]);
        }
        printf("%d ",queue[i]);
    }
}
```


Deque (Double Ended Queues)

- Insertion and Deletion are performed from both the ends, i.e.,
  - we can insert/delete elements from the **REAR** end or from the **FRONT** end
- Four operations are performed:
  - Insertion of an element at the **REAR** end of Queue.
  - Deletion of an element from the **FRONT** end of Queue.
  - Insertion of an element at the **FRONT** end of Queue.
  - Deletion of an element from the **REAR** end of Queue.
- There are two types of Deques:
  - **Input-restricted Deque**: Deletion can be performed from both ends (**FRONT** and **REAR**) while Insertion can be done at one end (**REAR**)
  - **Output-restricted Deque**: Deletion can be performed from one end (FRONT) while Insertion can be done at both ends (**REAR** and **FRONT**)

Implementation of Deque using Array

- Methods to be implemented for Deque

```
int isEmpty()
int isFull()
void insertFront(int x)
void insertRear(int x)
int deleteFront()
int deleteRear()
```


Bibliography

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"
- http://www.btechsmartclass.com/ (Online)

